Abstract:

This document, written by Zain Syed, the VP Tech at WE AutoPilot, will guide you through building (potentially) your first Machine Learning model! We will be using the MNIST dataset to make an AI tool to recognize handwritten numbers. The principle is, by submitting thousands of handwritten images, we can teach our computer to recognize patterns, making it capable of recognizing numbers. **WHY CARE?** Well, although it may seem like a trivial task to read handwritten numbers (unless it's my handwriting), for a computer it is really huge. If I get a calculator, or a computer to type out the number "8", they do not actually understand what 8 is, or what it represents. It is merely a symbol among others to a computer. If we can train a computer to **visually interpret and understand** a number, that is incredible news! It is like the natural next step when it comes to computer programming. Although this machine learning is only used to identify handwritten numbers, we can take it one step further, and do all sorts of incredible things with AI! Such as autonomous driving. *Check out this repo for more details.*
PLEASE REFER TO PAGE 12 FOR DICTIONARY, AND A HELPFUL LIST OF TERMS

Keep in mind: All commands in this color are for the terminal
All commands in this color are for our actual files

Part 1) Creating our model.py file:

Step 1) Installing modules and defining variables:

Our first step is to open up vscode and create a file which will allow us to start defining the model itself. In theory we could create our entire model, including the training and the GUI aspect in one file, but as computer programmers it is also our responsibility to make our code tidy and easy to read. By dividing up our files by the function they provide, we can make future upgrades and debugging much easier. We will start by creating a file on our current working directory (CWD) called ***model.py***

When we say defining the model, we will NOT be training yet. In PyTorch, the general format is to create the skeleton and design of the model first, and then to actually train that model with our dataset later (which will be the basis of our next file, train.py). In short, model.py will lay out all the groundwork for our model, and define **how** it will be trained, but the actual training will be done in another file.

As soon as model.py is created, we can start downloading and importing our required libraries. Firstly, we can install our sole dependency for this step

```
pip install torch
```

Then we can add the following module and it's submodules with the following code

```
import torch
from torch import nn
from torch.nn import functional
```

Simply put, these libraries/modules enable us to access methods that we initially would not be able to use. These functions will be integral, and really help us when it comes to defining our Convolutional Neural Network.

Next, we can start configuring some general variables, these variables will come in handy when defining exactly how we plan to train our model. Add the following lines to your code.

```
EPOCHS = 4
LEARNING_RATE = 0.001
GAMMA = 0.7
TRAINING_LOG_INTERVAL = 10_000
EPOCH_BREAK_ACCURACY = 0.995
TEST_BATCH_SIZE = 1000
```

Epochs will represent *how many times* we will iterate through our dataset in the training process. Learning rate is a value that determines how fast the model will learn information, we need to maximize our learning rate without making the learning process unstable, a high LR risks overshooting the optimal solution. Gamma is a value we use to decay our LR over time. Training Log Interval is solely for how often we log/document updates on the training. Epoch Break Accuracy is a threshold, training stops when this threshold is met. Test Batch Size represents how many images we will be testing at a time, images are grouped into batches which are inputted into the model together

Step 2) Creating our CNN class, and child functions:

Now, it is time for us to define the basic structure and data flow of our Convolutional Neural Network. This way, our computers know exactly what to do with data when it is sent to them. We can start by creating a class called CNN, which can store all our relevant functions.

```
class CNN(nn.Module):
    def __init__(self):
    def forward(self, x):
```

Here we have defined the Class CNN to encapsulate these two methods. The Initialization method will be where we create layers of our model and define their numeric values, the forward function will be used to detail how data will flow through our model, along with how we can convert data into something our model can use & learn from

Now, we can define our __init__ function, meaning this is where we start defining our neural network layers. This video, is an excellent starting point as to choosing your layers, for now, we can use these layers in our code: (Note that this code will simply fall under the function we defined in the last step). ***Check the slides to review WHY we chose the layers we did***

```
def __init__(self):
    super(CNN, self).__init__()
    self.conv1 = nn.Conv2d(1, 32, 3, 1) # Conv2d(in, out, kernel, stride)
    self.conv2 = nn.Conv2d(32, 64, 3, 1)
    self.dropout1 = nn.Dropout(0.25) # Dropout(percentOfNeurons)
    self.dropout2 = nn.Dropout(0.5)
    self.fc1 = nn.Linear(9216, 128)  # Linear(in, out)
```

Notice how we explicitly defined 5 layers, although we wish it was this easy, we still have more layers to add! Some layers can directly be added into our forwarding function, but some layers require some initialization. Some of our layers have initial states that must be explicitly declared, avoiding complicated logic errors. It may seem repetitive, but it makes our lives easier.

Since our important layers are defined and initialized, we can start coding our forward function, allowing us to define the data flow logic in our machine learning model! (How fun!)

```
def forward(self, x):
    x = self.conv1(x)
    x = functional.relu(x)
    x = self.conv2(x)
    x = functional.relu(x)
    x = functional.max_pool2d(x, 2)
    x = self.dropout1(x)
    x = torch.flatten(x, 1)
    x = self.fc1(x)
    x = functional.relu(x)
    x = self.dropout2(x)
    x = self.fc2(x)
    return x
```

This function is perhaps our most important, it defines exactly how data will be processed and extracted from an image that travels through our training algorithm. Below are definitions of each kind of layer, and exactly what their purpose is.

Convolutional Layers (conv1/2): These are the meat and potatoes, the job of these layers is to create filters (more formally known as kernels), in the form of n x n matrices. Each kernel *strides* over the pixels on our image, and looks for specific patterns. Each kernel on an image creates a new output to be sent to the next layers

Rectified Linear Unit (ReLU): An activation function which adds non-linearity into our model. In incredibly simple terms, it *rectifies* the input by turning ALL negative values into 0. The Convolutional layers will output positive and negative numbers, by using ReLU we remove unnecessary (negative) values, only keeping the strongest activations

Max Pooling Layer (max_pool2d): Very simply reduces the size of the input image(s) it receives, in the bracket above we can see the number 2, this simply means we reduce the size of the image by a factor of 2 pixels. Each 2x2 pixel grid is converted into 1 pixel value

Dropout Layer (dropout1/2): We temporarily disable a percentage of neurons in our neural network. The purpose of dropout layers is to reduce training time and prevent overfitting (Data being trained TOO well, where it memorizes patterns instead of recognizing simple ones)

Flatten Layer: A simple layer which converts multidimensional inputs into a 1 dimensional vector before passing it into the Fully Connected Layers.

Fully Connected Layers: Reduces the amount of neurons we have in our model, it does this by connecting EVERY input neuron to EVERY output neutron. This effectively converts our logical neurons into a smaller number of equally functional neurons.

Here is a table to make the visualization of what our layers are doing a little simpler:
KEEP IN MIND, our input is (28, 28, 1) because we input one 28x28 px image

Effects of Each Sequential Layer (in order)			
Layer Type	**Filters/Units**	**Activation**	**Output Shape**
Input	-	-	(28, 28, 1)
Conv2D	32 (5x5)	ReLU	(28, 28, 32)
Conv2D	64(5x5)	ReLU	(28, 28, 64)
MaxPooling2D	-	-	(14, 14, 64)
Dropout	25%	-	(14, 14, 64)
Flatten	-	-	(12,544) (Neurons)
Dense (Fully Connected)	128	ReLU	(128) (Neurons)
Dropout	50%	-	(128) (Neurons)
Dense (Fully Connected)	10	-	(10) (Neurons)

Step 3) Creating the train_model function:

This is the function we would use to define the procedure behind our training, we can use the CNN infrastructure we had created before, and input data into it as we attempt to train it. Although we are defining it here, we actually will be running it from our train.py file. We can start by defining our function: **(I recommend going over the comments for this code on GitHub)**

```
def train_model(model, device, data_loader, loss_func, optimizer, num_epochs=EPOCHS):
    train_loss, train_acc = [], []
    …
```

After creating the function, we can start adding our for loops, which allow us to iterate through the dataset and perform tasks for each image that we access. Under the last line, add:

```
    for epoch in range(num_epochs):
        runningLoss = 0.0
        correct, total = 0, 0
        for images, labels, in data_loader:
            …
```

This code will run through the entire dataset, for however many epochs our hearts desire. We have it set to 4 at the moment, as defined at the beginning of our code. The next step is to start building the logic behind what happens to each image and each label during the training process. We can start by adding the following to our most-nested for loop, (make sure indentation remains consistent with the *for images, labels, in data_loader:* line):

```
…
images, labels = images.to(device), labels.to(device)
optimizer.zero_grad()
outputs = model(images
loss = loss_func(outputs, labels)
loss.backward()
optimizer.step()
runningLoss += loss.item()
_, predicted = outputs.max(1)
correct += predicted.eq(labels).sum().item()
total += labels.size(0)
…
```

The code above pretty much passes every single input image through our model, and then tracks our loss (loss being our distance from the correct answer) value and updates the weights of our model with our loss function. Keep in mind the code above runs over and over again until it is done going through the dataset, by going backwards one indent, we can enter the scope of each epoch, instead of the dataset. Move back one indent and add this code:

```
        …
        epoch_loss = runningLoss / len(data_loader)
        epoch_acc = correct / total
        train_loss.append(epoch_loss)
        train_acc.append(epoch_acc)
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, Accuracy:
        {epoch_acc:.4f}")

        if epoch_acc >= 0.995:
                print("Model has reached 99.5% accuracy, stopping training")
                break
    return train_loss, train_acc
```

The code above allows us to add loss and accuracy values to the previously defined train_loss & train_acc arrays. After that, we print out the relevant accuracy and loss values. It also has a break loop, where it will stop training (break out of the epochs early) in the off chance our model reaches 99.5% accuracy early. The final line is one indent backwards, it simply just returns the two arrays we defined (returns them to the location the function was called), and ends our train_model function.

If done correctly, the indentations should look like this, ensure your lines match up

```
def train_model(model, device, data_loader, loss_func, optimizer, num_epochs = EPOCHS):
    …
    for epoch in range(num_epochs):
        …
        for images, labels in data_loader:
            …
        print(...)
    return train_loss, train_acc
```

Step 4) Defining our test_model function:

Of course we have created a function for training, it is natural we create one to test the true accuracy of our model. It of course will be done with the MNIST training dataset, images provided by the MNIST dataset that were not used for training.

This method will kind of be a simpler version of the training function, all it will do is load up our CNN model, and just start testing itself based on new images.

This code will consist of changing to evaluation mode for the model, along with disabling gradient calculations. Our testing then should compile very quickly, where we can output our accuracy and test loss values. Add the following function to your code:
Once again, I highly recommend you check the code in the GitHub repository to fully understand how this code tests the model.

```
def test_model(model, data_loader, device=None):
    if device is None:
     device = torch.device('cpu')

    model.eval()
    test_loss = 0
    correct = 0

    data_len = len(data_loader.dataset)

    with torch.no_grad():
        for data, target in data_loader:
            data, target = data.to(device), target.to(device)
            output = model(data) # Retrieving predictions of our model
            test_loss += functional.cross_entropy(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(data_loader.dataset) #Calculating average loss
    accuracy = correct / data_len
    print(f'Test set: Average loss: {test_loss:.4f}, Accuracy: {correct}/{data_len} ({100 *
accuracy}%)')

    return accuracy
```

And with that, Part 1 is officially complete! The next steps are to make our train.py file, training our model, (a lot of waiting), and then implementing some codes for graph visualization, and whiteboard CNN testing! Once that is all said and done, we should be able to implement our code into a simple whiteboard GUI to start manually verifying its validity.

Part 2) Creating train.py

Step 1) Importing modules and defining variables

Similar to last time, we can do some basic lines of code to gather the relevant libraries, as well as setting up some values which would be useful in this file.

```
import torch
from torch.utils.data import DataLoader
from torch.optim import RMSprop
from torchvision import datasets
from model import CNN, train_model, test_model
from lib.util import header, plot_distribution, double_plot, normalization_transform

SESSION_1_EPOCH_COUNT = 3
SESSION_2_EPOCH_COUNT = 3
LEARNING_RATE = 0.001
BATCH_SIZE = 64
TEST_BATCH_SIZE = 1000
```

We are importing some public modules, as well as the code we wrote in the model.py file previously. We also are importing from a file called util.py in the lin folder, but this tutorial will not cover that, the code will be provided to you! We also defined some necessary variables that will come into play later.

The following code will be used for choosing the best possible device for training, the possibilities are GPU, then MPS, and last resort is the standard CPU.

```
def det_device_config(train_kwargs, test_kwargs):
    use_cuda = torch.cuda.is_available()
    use_mps = torch.backends.mps.is_available()
    if use_cuda:
        print('using cuda device')
        device = torch.device("cuda")
    elif use_mps:
        print('using mps')
        device = torch.device("mps")
    else:
        print('using cpu')
        device = torch.device("cpu")
    if use_cuda:
        cuda_kwargs = {'num_workers': 1,
                       'pin_memory': True,
                       'shuffle': True}
        train_kwargs.update(cuda_kwargs)
        test_kwargs.update(cuda_kwargs)
    return device, train_kwargs, test_kwargs
```

Step 2) Training the model (+ How to save training with checkpoints)

The next thing we are going to do is start working on the rest of our code for train.py, we can define an if statement to encapsulate the rest of our code. Create this if statement, and indent once on the next line (should happen automatically on VSCode). ***Make sure you keep this indent consistent for the rest of the code.***

```
If __name__ = ‘__main__’:
```

Now we can start loading the functions from model.py and start training, add the following code under the if statements, it will initialize dictionaries to store arguments, and then create our model and attach it to the current device

```
train_kwargs = {‘batch_size’: BATCH_SIZE}
test_kwargs = {‘batch_size’: TEST_BATCH_SIZE}
device, train_kwargs, test_kwargs = det_device_config(train_kwargs, test_kwargs)
Model = CNN().to(device)
```

Next, we can download the MNIST dataset and normalize the data to fit our needs.

```
header("Loading datasets...")
train_data = datasets.MNIST('../data', train=True, download=True, transform=normalization_transform)
test_data = datasets.MNIST('../data', train=False, transform=normalization_transform)
print("Done loading datasets")
```

When using unfamiliar datasets it is great to visualize our data to make sure the distribution seems right. The following code takes all the labels and plots them on a bar graph to get a sense of how commonly each number occurs. Look at the data (when we run the code) and identify the numbers that occur the most, and then the numbers that occur the least. Why do you think the dataset was made this way? [If you do not want to add this, check out the graph]

```
plot_distribution('Distribution of Labels in Training Set', train_data)
plot_distribution('Distribution of Labels in Testing Set', test_data)
```

After plotting our data, we can start actually training our model. The way we plan on doing it in this workshop is by training one session, stopping to save as a checkpoint, loading the checkpoint, and continuing our training from there. In a general sense we wouldn’t do it in this situation, but many times it is a great idea to save a checkpoint of the trained model, especially if we plan on coming back to it later on. Here is the code for our first training session

```
header("Training the model...")
optimizer = RMSprop(model.parameters(), lr=LEARNING_RATE)
train_loss, train_acc = train_model(
        model,
        device,
        data_loader=DataLoader(train_data,**train_kwargs),
        loss_func=torch.nn.CrossEntropyLoss(),
        optimizer=optimizer,
        num_epochs=SESSION_1_EPOCH_COUNT
)
print("Done training")
```

After that, we can save our code as a checkpoint, this part is optional however it is a really good thing to know (keep in mind, if you decide to omit the checkpoint step, some changes will have to be made to the code to train everything in one go)

```
# save the checkpoint
header('Saving checkpoint 1...')
checkpoint_1_path = "../checkpoint1.pt"
torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
}, checkpoint_1_path)
print("Saved")

# loading the checkpoint
header('Loading checkpoint 1...')
checkpoint = torch.load(checkpoint_1_path)
new_model = CNN().to(device)
new_model.load_state_dict(checkpoint['model_state_dict'])
new_optimizer = RMSprop(new_model.parameters(), lr=LEARNING_RATE)
new_optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
print("Loaded")
```

Another good practice is to check our accuracy before we start training again, we can do that with the following code

```
print("Checking Accuracy...")
old_test_accuracy = test_model(model, DataLoader(test_data, **test_kwargs), device)
new_test_accuracy = test_model(new_model, DataLoader(test_data, **test_kwargs), device)
print("Loaded Accuracy: ", new_test_accuracy)
print("Expected Accuracy: ", old_test_accuracy)
print("Done")
```

The next step is naturally to go back to training our model after loading the checkpoint backup.

```
header("Training the model...")
train_loss_2, train_acc_2 = train_model(
    new_model,
    device,
    data_loader=DataLoader(train_data,**train_kwargs),
    loss_func=torch.nn.CrossEntropyLoss(),
    optimizer=new_optimizer,
    num_epochs=SESSION_2_EPOCH_COUNT
)
print('Done training')
```

Use the following code to combine the 2 training sessions, which will complete our training

```
train_loss.extend(train_loss_2)
train_acc.extend(train_acc_2)
```

Step 3) Saving the model and plotting loss and accuracy charts

The final, and perhaps most simple step in this part, is to save our model as a .pt file, and then plot our data. We can do the following with these last 5 lines of code. Make sure to keep this all in the same indent we defined for the if statement in the beginning of Step 2.

```
header("Saving the model to file...")
MODEL_PATH = "../mnist_cnn.pt"
torch.save(new_model.state_dict(), MODEL_PATH)
print(f"Saved model to {MODEL_PATH}")
double_plot(label1="Training Loss", data1=train_loss, label2="Training Accuracy", data2=train_acc)
```

Part 3) Adding library folder, and 2 provided files

This next part is less about the AI model itself, and more about some other tools, and the GUI that we have provided. We won't teach you guys about this stuff, instead it is readily available on the GitHub Repository.

Open up the directory, and open this folder:
MNIST > src > lib
Copy the entirety of the contents of this lib folder (including the folder itself) and move it to the same directory (folder) where you are saving train.py and model.py. It is important that those 2 files are in the same directory as this lib folder for the code to successfully work.

Part 4) Running our code

The first thing we need to do is execute our train.py file, since it imports the code from model.py, that will not need to be run (it's a good practice to split our files up like that to maintain code cleanliness and readability). We can run train.py by opening up the terminal, and changing directories to the same CWD as the train.py file (feel free to google this or better yet ask a mentor for help if you are not comfortable with it).

After you are in the correct directory, run this command in the terminal

```
python train.py
```

Make sure the entire code compiles without any errors, don't worry if it takes a long time, remember we are running 60,000 images through the training model (6 times), and then 10,000 images through testing.

Once it is done, we can see the accuracy of the models in the same terminal we ran the command in (also you will see some graphs pop up, they are cool to look at, but make sure to close the graphs when they do pop up, because the code stops and waits for that process to end before it continues on to the next parts in the execution process).

To test your model, our incredible president Aly Ashour has set up a simple GUI, run these commands: Here, you can draw numbers and see your model (mostly) work in real time!

```
cd lib
Python gui.py
```

EXTRA CHALLENGES:

- Find where we would define/change the learning rate, and see what changes you can make there to potentially make training faster or more stable. HINT: Learning rate is not explicitly defined in my code, it uses the default value but not declaring it in the code, see how changing this value can affect training your model. Try doing the same thing but with Epochs, see how changing the number of Epochs can affect the training stage of the model.
- Check out Tensorflow if you are interested, (I definitely prefer it over PyTorch), there are some great guides and implementations in this repository I worked on (before the workshop was switched to PyTorch). Try training MNIST with Tensorflow and identify any potential differences between it and PyTorch
- See if you can implement the GUI that Aly provided into some sort of website or web application, you can make a cool place to show off your excellent work!
- Try and teach someone else about Neural Networks, and how to code using PyTorch. You may not be a professional just yet, but teaching is an excellent way to solidify your learning. You still have this document, and the public repository to help you out!

FAQs and Common Problems:

Error: Could not install packages due to OSError:[WinError 2], system cannot find file specified
This error can easily be solved, it likely means Python isn't installed, or the path is not correct.
To solve it: Hit **Win+R**, and type **sysdm.cpl**, then navigate to:
Advanced > Environment Variables > User variables for [username] > Path
Now you can click **New**, and add the following to your list of Paths,
C:\Python310\
C:\Python310\Scripts
Be careful before you add these to path, you may have another version of python installed, let a mentor double check everything before you do it.
After the path is added, restart your computer, open a terminal and type where python it should return some path, which means you did it correctly, and the error shouldn't persist

Dictionary:

General Terminology:

Machine Learning (ML): A field of AI where computers learn from data instead of being explicitly programmed.
Neural Network (NN): A model inspired by the human brain that consists of layers of neurons that process data.
Deep Learning: A subset of ML where multi-layered neural networks are used to automatically learn patterns from data.
Training Data: The dataset used to teach a model by adjusting its parameters based on patterns found in examples.
Test Data: A separate dataset used to evaluate a trained model's performance on unseen data.
Overfitting: A problem where a model learns the training data too well, including noise, making it bad at generalizing to new data.
Underfitting: A problem where a model is too simple to capture important patterns in the training data.
Epoch: A complete pass through the entire training dataset. More epochs allow the model to improve its understanding of patterns.
Batch Size: The number of samples used in one update step of training. Smaller batch sizes lead to more updates per epoch, while larger batch sizes speed up training.
Gradient Descent: An optimization algorithm that adjusts the model's parameters to minimize the loss function.
Backpropagation: The process of updating model parameters by computing gradients using the chain rule of differentiation.

Convolutional Neural Networks:

Convolutional Layer (Conv2D): A layer in CNNs that detects patterns (like edges and textures) in an image using filters/kernels.
Filter/Kernels: Small matrices that scan over an image, extracting important spatial features.
Stride: The number of pixels a filter moves at each step while scanning the image.
Activation Function: A function applied to neurons to introduce non-linearity, making the network capable of learning complex patterns.
ReLU (Rectified Linear Unit): The most common activation function, setting negative values to 0 while keeping positive values.
Max Pooling: A layer that reduces image size while keeping important features by selecting the max value in each region of pixels.
Flatten Layer: Converts the 2D feature maps into a 1D vector so they can be used in fully connected layers.
Fully Connected Layer (FC or Dense Layer): A layer where every neuron is connected to every neuron in the previous layer—often used at the end of CNNs for classification.
Dropout Layer: A regularization technique where random neurons are "turned off" during training to prevent overfitting.

Model Training and Loss Functions

Loss Function: Measures how far the model's predictions are from the actual labels.
Cross-Entropy Loss: A loss function commonly used for classification tasks—it compares predicted probabilities with the correct labels.
NLL (Negative Log Likelihood) Loss: Used with log_softmax(), it penalizes incorrect predictions based on confidence.
Optimizer: An algorithm that updates the model's weights to minimize the loss.
RMSprop (Root Mean Square Propagation): An adaptive learning rate optimizer that helps prevent oscillations in weight updates.
Adam (Adaptive Moment Estimation): A commonly used optimizer that adapts the learning rate based on past gradients.

PyTorch Terminology:

Tensor: The fundamental data structure in PyTorch, similar to a NumPy array but optimized for GPU operations.
torch.nn.Module: The base class for all PyTorch models, used to define architectures.
torch.nn.functional: A module containing useful functions like activation layers, loss functions,
torch.optim: A module containing optimizers like SGD, Adam, and RMSprop.
torch.cuda: A module for moving tensors and models to the GPU for faster computation.
State Dict: A dictionary storing all model parameters (weights and biases), which is used when saving/loading models.

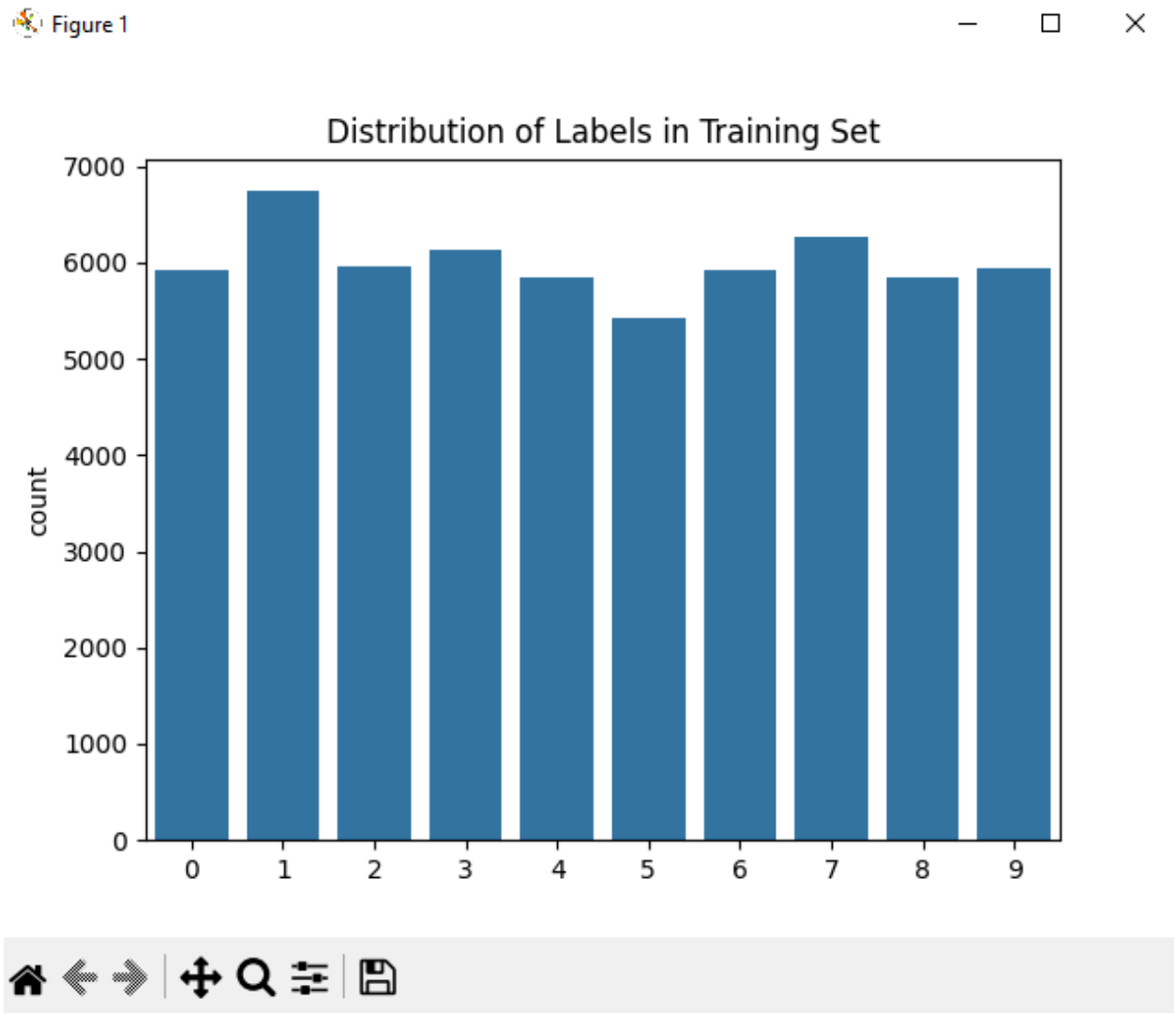
Preprocessing & Data Handling:

DataLoader: A PyTorch utility that loads datasets in batches, making training more efficient.
Transformations: Operations applied to datasets before training, such as normalization and augmentation.
Normalization: Adjusting pixel values so they have zero mean and unit variance, helping models learn faster.
ToTensor(): Converts an image from PIL/Numpy format into a PyTorch tensor.

Saving, Loading & Deployment:

Checkpoint: A saved state of the model during training, allowing you to resume training from the same point.
torch.save(): Saves a model's state_dict (weights & biases).
torch.load(): Loads a saved model's state_dict for later use.
.pt vs .pth Files: Both file extensions are used for saving PyTorch models. .pt is preferred for sharing models, while .pth is often used for internal development.
ONNX (Open Neural Network Exchange): A format that allows PyTorch models to be exported and used in other frameworks (like TensorFlow or C++ apps).

Distribution of numbers in MNIST Dataset



Look at the distribution of labels in this dataset, they are relatively evenly distributed, but the question is, ***why not perfectly distributed?*** It is a valid question, it is a bunch of handwritten numbers, why couldn't the creators of this dataset be distributed in a balanced manner?

There actually is more to this question than you may think, take a look at numbers that have more appearances than others, and try and guess why they would be distributed this way.
Hint: Is there a reason one number may need to be trained more than another? Why?

Now look at the numbers with lower distribution, can you come up with a reason as to why they would be less prevalent in the training set?
Hint: Is there a reason one number may need to be trained less than another? Why?